**RESTful Web Services Testing**

**V1.0**

# **What is Rest API?**

**Representational State Transfer** (**REST**) is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, termed *RESTful* Web services (RWS), provide interoperability between computer systems on the Internet. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations. Other kinds of Web services, such as SOAP Web services, expose their own arbitrary sets of operations.

In a RESTful Web service, requests made to a resource's URI will elicit a response with a payload formatted in HTML, XML, JSON, or some other format. The response can confirm that some alteration has been made to the stored resource, and the response can provide hypertext links to other related resources or collections of resources. When HTTP is used, as is most common, the operations (HTTP methods) available are GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS and TRACE.

By using a stateless protocol and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow, by re-using components that can be managed and updated without affecting the system as a whole, even while it is running.

**Properties of RESTful Web services:**

* + simplicity of a uniform interface
  + modifiability of components to meet changing needs (even while the application is running);
  + visibility of communication between components by service agents;
  + portability of components by moving program code with the data;
  + reliability in the resistance to failure at the system level in the presence of failures within components, connectors, or data
  + scalability allowing the support of large numbers of components and interactions among components.

# **Architectural Constraints:**

Six guiding constraints define a RESTful system. These constraints restrict the ways that the server can process and respond to client requests so that, by operating within these constraints, the system gains desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.If a system violates any of the required constraints, it cannot be considered RESTful.

The formal REST constraints are as follows:

**Client–server architecture**

The principle behind the client–server constraints is the separation of concerns. Separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

**Statelessness**

The client–server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be *in transition*. The representation of each application state contains links that can be used the next time the client chooses to initiate a new state-transition.

**Cacheability**

As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from getting stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

**Layered system**

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches. They can also enforce security policies.

**Code on demand (optional)**

Servers can temporarily extend or customize the functionality of a client by transferring executable code: for example, compiled components such as Java applets, or client-side scripts such as JavaScript.

### Uniform interface

The uniform interface constraint is fundamental to the design of any RESTful system. It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

**Resource identification in requests**

Individual resources are identified in requests, for example using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server could send data from its database as HTML, [XML](https://en.wikipedia.org/wiki/XML) or as JSON—none of which are the server's internal representation.

**Resource manipulation through representations**

When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.

**Self-descriptive messages**

Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.

**Hypermedia as the engine of application state (HATEOAS)**

Having accessed an initial URI for the REST application—analogous to a human Web user accessing the [home page](https://en.wikipedia.org/wiki/Home_page) of a website—a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs. As access proceeds, the server responds with text that includes [hyperlinks](https://en.wikipedia.org/wiki/Hyperlink) to other actions that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the application.

# **HTTP Methods:**

The HTTP verbs comprise a major portion of our “uniform interface” constraint and provide us the action counterpart to the noun-based resource. The primary or most-commonly-used HTTP verbs (or methods, as they are properly called) are POST, GET, PUT, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations, respectively. There are a number of other verbs, too, but are utilized less frequently. Of those less-frequent methods, OPTIONS and HEAD are used more often than others.

### HTTP GET

Use GET requests **to retrieve resource representation/information only** – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be **safe methods**. Additionally, GET APIs should be **idempotent**, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

For any given HTTP GET API, if the resource is found on the server then it must return HTTP response code 200 (OK) – along with response body which is usually either XML or JSON content (due to their platform independent nature).

In case resource is NOT found on server then it must return HTTP response code 404 (NOT FOUND). Similarly, if it is determined that GET request itself is not correctly formed then server will return HTTP response code 400 (BAD REQUEST).

Example request URIs

HTTP GET http://www.appdomain.com/users

HTTP GET http://www.appdomain.com/users?size=20&page=5

HTTP GET http://www.appdomain.com/users/123

HTTP GET http://www.appdomain.com/users/123/address

### HTTP POST

Use POST APIs **to create new subordinate resources**, e.g. a file is subordinate to a directory containing it or a row is subordinate to a database table. Talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources.

Ideally, if a resource has been created on the origin server, the response SHOULD be HTTP response code 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header.

Many times, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either HTTP response code 200 (OK) or 204 (No Content) is the appropriate response status.

Responses to this method are **not cacheable**, unless the response includes appropriate Cache-Control or Expires header fields.

Please note that POST is **neither safe nor idempotent** and invoking two identical POST requests will result in two different resources containing the same information (except resource ids).

Example request URIs

HTTP POST http://www.appdomain.com/users

HTTP POST <http://www.appdomain.com/users/123/accounts>

### HTTP PUT

Use PUT APIs primarily **to update existing resource** (if the resource does not exist then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are **not cacheable**.

The difference between the POST and PUT APIs can be observed in request URIs. POST requests are made on resource collections whereas PUT requests are made on an individual resource.

Example request URIs

HTTP PUT http://www.appdomain.com/users/123

HTTP PUT http://www.appdomain.com/users/123/accounts/456

### HTTP DELETE

As the name applies, DELETE APIs are used **to delete resources** (identified by the Request-URI).

A successful response of DELETE requests SHOULD be HTTP response code 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action has been queued, or 204 (No Content) if the action has been performed but the response does not include an entity.

DELETE operations are **idempotent**. If we DELETE a resource, it’s removed from the collection of resource. Repeatedly calling DELETE API on that resource will not change the outcome – however calling DELETE on a resource a second time will return a 404 (NOT FOUND) since it was already removed. Some may argue that it makes DELETE method non-idempotent. It’s a matter of discussion and personal opinion.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are **not cacheable**.

Example request URIs

HTTP DELETE http://www.appdomain.com/users/123

HTTP DELETE <http://www.appdomain.com/users/123/accounts/456>

| **HTTP Method** | **CRUD** | **Entire Collection (e.g. /users)** | **Specific Item (e.g. /users/123)** |
| --- | --- | --- | --- |
| POST | Create | 201 (Created), ‘Location’ header with link to /users/{id} containing new ID. | Avoid using POST on single resource |
| GET | Read | 200 (OK), list of users. Use pagination, sorting and filtering to navigate big lists. | 200 (OK), single user. 404 (Not Found), if ID not found or invalid. |
| PUT | Update/Replace | 404 (Not Found), unless we want to update every resource in the entire collection of resource. | 200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid. |
| DELETE | Delete | 404 (Not Found), unless we want to delete the whole collection — use with caution. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

# **HTTP Status Codes**

REST APIs use the **Status-Line** part of an HTTP response message to inform clients of their request’s overarching result.

HTTP defines forty standard status codes that can be used to convey the results of a client’s request. The status codes are divided into the five categories presented below.

|  |  |
| --- | --- |
| **Category** | **Description** |
| **1xx: Informational** | Communicates transfer protocol-level information. |
| **2xx: Success** | Indicates that the client’s request was accepted successfully. |
| **3xx: Redirection** | Indicates that the client must take some additional action in order to complete their request. |
| **4xx: Client Error** | This category of error status codes points the finger at clients. |
| **5xx: Server Error** | The server takes responsibility for these error status codes. |

#### 200 (OK)

It indicates that the REST API successfully carried out whatever action the client requested, and that no more specific code in the 2xx series is appropriate.

Unlike the 204 status code, a 200 response should include a response body. The information returned with the response is dependent on the method used in the request, for example:

* GET an entity corresponding to the requested resource is sent in the response;
* HEAD the entity-header fields corresponding to the requested resource are sent in the response without any message-body;
* POST an entity describing or containing the result of the action;
* TRACE an entity containing the request message as received by the end server.

#### 201 (Created)

A REST API responds with the 201 status code whenever a resource is created inside a collection. There may also be times when a new resource is created as a result of some controller action, in which case 201 would also be an appropriate response.

The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URI for the resource given by a Location header field.

The origin server MUST create the resource before returning the 201 status code. If the action cannot be carried out immediately, the server SHOULD respond with 202 (Accepted) response instead.

#### 202 (Accepted)

A 202 response is typically used for actions that take a long while to process. It indicates that the request has been accepted for processing, but the processing has not been completed. The request might or might not be eventually acted upon, or even maybe disallowed when processing occurs.

Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent’s connection to the server persist until the process is completed.

The entity returned with this response SHOULD include an indication of the request’s current status and either a pointer to a status monitor (job queue location) or some estimate of when the user can expect the request to be fulfilled.

#### 204 (No Content)

The 204 status code is usually sent out in response to a PUT, POST, or DELETE request when the REST API declines to send back any status message or representation in the response message’s body.

An API may also send 204 in conjunction with a GET request to indicate that the requested resource exists, but has no state representation to include in the body.

If the client is a user agent, it SHOULD NOT change its document view from that which caused the request to be sent. This response is primarily intended to allow input for actions to take place without causing a change to the user agent’s active document view, although any new or updated meta-information SHOULD be applied to the document currently in the user agent’s active view.

The 204 response MUST NOT include a message-body and thus is always terminated by the first empty line after the header fields.

#### 301 (Moved Permanently)

The 301 status code indicates that the REST API’s resource model has been significantly redesigned and a new permanent URI has been assigned to the client’s requested resource. The REST API should specify the new URI in the response’s Location header and all future requests should be directed to the given URI.

We will hardly use this response code in our API as we can always use the API versioning for new API while retaining the old one.

#### 302 (Found)

The HTTP response status code 302 Found is a common way of performing URL redirection. An HTTP response with this status code will additionally provide a URL in the location header field. The user agent (e.g. a web browser) is invited by a response with this code to make a second, otherwise identical, request to the new URL specified in the location field.

Many web browsers implemented this code in a manner that violated this standard, changing the request type of the new request to GET, regardless of the type employed in the original request (e.g. POST). RFC 1945 and RFC 2068 specify that the client is not allowed to change the method on the redirected request. The status codes 303 and 307 have been added for servers that wish to make unambiguously clear which kind of reaction is expected of the client.

#### 303 (See Other)

A 303 response indicates that a controller resource has finished its work, but instead of sending a potentially unwanted response body, it sends the client the URI of a response resource. This can be the URI of a temporary status message, or the URI to some already existing, more permanent, resource.

Generally speaking, the 303 status code allows a REST API to send a reference to a resource without forcing the client to download its state. Instead, the client may send a GET request to the value of the Location header.

The 303 response MUST NOT be cached, but the response to the second (redirected) request might be cacheable.

#### 304 (Not Modified)

This status code is similar to 204 (“No Content”) in that the response body must be empty. The key distinction is that 204 is used when there is nothing to send in the body, whereas 304 is used when the resource has not been modified since the version specified by the request headers If-Modified-Since or If-None-Match.

In such case, there is no need to retransmit the resource since the client still has a previously-downloaded copy.

Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

#### 307 (Temporary Redirect)

A 307 response indicates that the REST API is not going to process the client’s request. Instead, the client should resubmit the request to the URI specified by the response message’s Location header. However, future requests should still use the original URI.

A REST API can use this status code to assign a temporary URI to the client’s requested resource. For example, a 307 response can be used to shift a client request over to another host.

The temporary URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s). If the 307 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

#### 400 (Bad Request)

400 is the generic client-side error status, used when no other 4xx error code is appropriate. Errors can be like malformed request syntax, invalid request message parameters, or deceptive request routing etc.

The client SHOULD NOT repeat the request without modifications.

#### 401 (Unauthorized)

A 401 error response indicates that the client tried to operate on a protected resource without providing the proper authorization. It may have provided the wrong credentials or none at all. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource.

The client MAY repeat the request with a suitable Authorization header field. If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user SHOULD be presented the entity that was given in the response, since that entity might include relevant diagnostic information.

#### 403 (Forbidden)

A 403 error response indicates that the client’s request is formed correctly, but the REST API refuses to honor it i.e. the user does not have the necessary permissions for the resource. A 403 response is not a case of insufficient client credentials; that would be 401 (“Unauthorized”).

Authentication will not help and the request SHOULD NOT be repeated. Unlike a 401 Unauthorized response, authenticating will make no difference.

#### 404 (Not Found)

The 404 error status code indicates that the REST API can’t map the client’s URI to a resource but may be available in the future. Subsequent requests by the client are permissible.

No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

#### 405 (Method Not Allowed)

The API responds with a 405 error to indicate that the client tried to use an HTTP method that the resource does not allow. For instance, a read-only resource could support only GET and HEAD, while a controller resource might allow GET and POST, but not PUT or DELETE.

A 405 response must include the Allow header, which lists the HTTP methods that the resource supports. For example:

Allow: GET, POST

#### 406 (Not Acceptable)

The 406 error response indicates that the API is not able to generate any of the client’s preferred media types, as indicated by the Accept request header. For example, a client request for data formatted as application/xml will receive a 406 response if the API is only willing to format data as application/json.

If the response could be unacceptable, a user agent SHOULD temporarily stop receipt of more data and query the user for a decision on further actions.

#### 412 (Precondition Failed)

The 412 error response indicates that the client specified one or more preconditions in its request headers, effectively telling the REST API to carry out its request only if certain conditions were met. A 412 response indicates that those conditions were not met, so instead of carrying out the request, the API sends this status code.

#### 415 (Unsupported Media Type)

The 415 error response indicates that the API is not able to process the client’s supplied media type, as indicated by the Content-Type request header. For example, a client request including data formatted as application/xml will receive a 415 response if the API is only willing to process data formatted as application/json.

For example, the client uploads an image as image/svg+xml, but the server requires that images use a different format.

#### 500 (Internal Server Error)

500 is the generic REST API error response. Most web frameworks automatically respond with this response status code whenever they execute some request handler code that raises an exception.

A 500 error is never the client’s fault and therefore it is reasonable for the client to retry the exact same request that triggered this response, and hope to get a different response.

API response is the generic error message, given when an unexpected condition was encountered and no more specific message is suitable.

#### 501 (Not Implemented)

The server either does not recognize the request method, or it lacks the ability to fulfill the request. Usually, this implies future availability (e.g., a new feature of a web-service API).

# **RESTful Web-services Testing**

Now days with increase in agility and devops, the testing is not only restricted to UI and databases. To provide better quality of code and software Web services testing is important. This increase the confidence of the developers as well the Business Analyst that the functionality of the software is as per the business requirement. To test this Web services, the tester must have through knowledge of technical as well as the business aspect of the software. To facilitate and reduce coding of internal working of Web services various framework are available in open-source market. Some of most widely framework/libraries are discussed below.

### Rest Assured Framework:

REST Assured is a Java DSL for simplifying testing of REST based services built on top of HTTP Builder. It supports POST, GET, PUT, DELETE, OPTIONS, PATCH and HEAD requests and can be used to validate and verify the response of these requests.

REST Assured can be used to test XML as well as JSON based web services. REST Assured can be integrated with Junit, TestNG and Cucumber frameworks for writing test cases for the application.

One of the powerful features of REST assured is the support of XML Path and JSON Path syntax to check specific elements of the response data. It’s very similar to using XPath API.

It uses Hamcrest API for matching the actual data with expected data.

Below is the link to get started with Rest Assured Framework.

<http://rest-assured.io/>

### Rest Template:

This is a framework provided by Spring team. It lets us to design REST APIs as well the feasibility to test them. Together with JSON Assert package it is a powerful testing tool for automating REST services, backed with Spring MVC framework.

One of the advantage of Rest template is it can be integrated with any testing frameworks. Provides library for retrieving and testing the services.

Another advantage of Rest template is its conversion of JSON/XML objects directly to POJO. Rest template uses Jersey library for parsing the JSON data.

It also supports all the Spring annotation for ease to load data at runtime. This framework helps in building clean and readable code due to the use of Spring IOC (Inversion of control/Dependency Injection) containers.

Below link will help in getting started with Rest template

<https://spring.io/guides/gs/consuming-rest/>

### Apache HTTPClient

Another famous API framework is HTTPClient developed by Apache team. It is built on top of java.net package. It provides wrapper methods and extension to java.net package.

Features of HTTPClient:

* Standards based, pure Java, implementation of HTTP versions 1.0 and 1.1
* Full implementation of all HTTP methods (GET, POST, PUT, DELETE, HEAD, OPTIONS, and TRACE) in an extensible OO framework.
* Supports encryption with HTTPS (HTTP over SSL) protocol.
* Transparent connections through HTTP proxies.
* Tunneled HTTPS connections through HTTP proxies, via the CONNECT method.
* Support for HTTP/1.1 response caching.
* Source code is freely available under the Apache License.

Below link will help with getting started with HTTPClient

<http://hc.apache.org/httpcomponents-client-ga/index.html>

### Java.Net.URLConnection Package

This is most basic and oldest but still in use framework directly available in java library. All the existing indirectly is built on top of this framework. All the operations on the web services can be performed using this library. URLConnection is an HTTP-centric class, that is most of the methods are helpful in working with HTTP URLs. Following the typical steps followed to communicate with server via URLs.

1. Create a URL object
2. Obtain a URLConnection object from the URL
3. Configure the URLConnection
4. Read the header fields
5. Get an input stream and read data
6. Get an output stream and write data
7. Close the connection

Below link will provide the methods and description of the URLConnection class

<https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html>